LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-SM-410820

Random Forest Toolbox: A Compact User's Guide

B. Chen, T. Lemmond, W. Hanley, J. Buyer, L. Hiller, D. Knapp

February 24, 2009

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Random Forest Toolbox: A Compact User's Guide

Lawrence Livermore National Laboratory[1]
Development Team:

Barry Chen
Tracy Lemmond
William Hanley
John Buyer
Lawrence Hiller
David Knapp

[1] This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory in part under Contract W-7405-Eng-48 and in part under Contract DE-AC52-07NA27344.

1. Installation and Compilation

The Random Forest (RF) Toolbox is a suite of programs (written in C++), Perl scripts, and sample data files to support the training, testing, and performance reporting of Random Forest-based classifiers. These inherently two-class classifiers operate on continuous or discrete input features and include:

- Random Forest with misclassification-based node splitting (RF)
- Random Forest with Gini Impurity-based node splitting (GRF)
- Cost-Sensitive Random Forest (CS-RF)
- Discriminant Random Forest (DRF)

Further details describing each of the supported classifiers can be found in the **Random Forest Variants** section of this User's Guide.

Prior to compiling the toolbox, the user should select a *base directory* where the source code will be located. For simplicity, throughout this document we will often refer to this directory generically as <toolbox_base>. Once a desired location has been selected, the user must copy the Toolbox archive to <toolbox_base> and unpack it.

1.1 Unix and Mac

Once unpacked, the toolbox supports two methods for compilation: 1) the Unix `make` utility (tested on Unix-like platforms PC Cygwin, PC Linux, and Mac OS X, with `g++`), or 2) Microsoft Visual Studio (for Windows platforms). To compile the toolbox using the `make` utility, simply navigate to the `<toolbox_base>` directory and type "`make`" at the prompt. This will compile the C++ code and create the toolbox's binary executables:

- `<toolbox_base>/Forests/Random/bin/train`[2]
- `<toolbox_base>/Forests/Random/bin/classify`
- `<toolbox_base>/Forests/Random/bin/proximity`
- `<toolbox_base>/Forests/Random/bin/forestcat`
- `<toolbox_base>/ComputeROC/bin/ComputeROC`
- `<toolbox_base>/ComputeCost/bin/ComputeCost`

The `Makefile` found in `<toolbox_base>` can also be used to clean up all intermediate compilation files and final binary executables. To perform this cleanup simply type "`make allclean`".

1.2 Windows

To build the toolbox executables using Microsoft Visual Studio 2005, open the "Forest" solution (`<toolbox_base>/Forests/msvc6/Forests.sln`), click the "Build" drop-down menu, and click "Batch Build". This will open up a "Batch Build" dialog box. On the right hand side, click the "Build" button to compile code. This creates the binary executables:

[2] In the case of compiling under Cygwin, the binary executables will have a "`.exe`" suffix.

- <toolbox_base>/Forests/msvc6/rndforest/Release/train.exe
- <toolbox_base>/Forests/msvc6/rndforest/Release/classify.exe
- <toolbox_base>/Forests/msvc6/rndforest/Release/proximity.exe
- <toolbox_base>/Forests/msvc6/rndforest/Release/forestcat.exe

Open the ComputeROC solution
(<toolbox_base>/ComputeROC/msvc6/ComputeROC.sln)
and ComputeCost solution
(<toolbox_base>/ComputeCost/msvc6/ROC.sln)
and repeat the "Batch Build" instructions above. The following executables will be built:

- <toolbox_base>/ComputeROC/msvc6/Release/ComputeROC.exe
- <toolbox_base>/ComputeCost/msvc6/Release/ComputeCost.exe

2. Executables and File Formats

train

`train` is the binary for training all of the Random Forest variants supported by this toolbox. Minimally, it reads at least one training file containing sample feature vectors and labels and outputs a Random Forest model file. Optionally, user-defined training parameters may be specified that determine numerous other Forest settings, described in detail in the ***train Parameters*** section. Figure 1 depicts the required (solid arrows) and optional (dashed arrows) inputs and outputs for `train`.

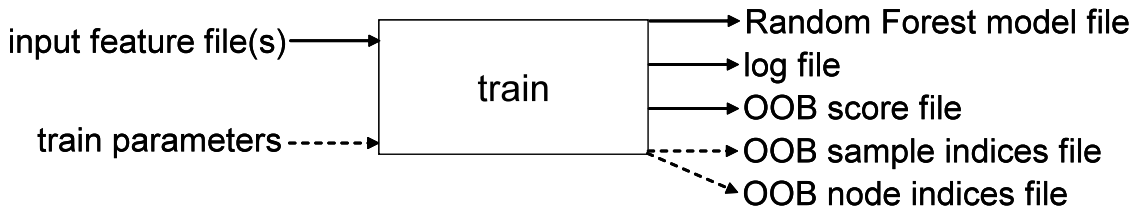

Figure 1 – Box diagram for the inputs and outputs of the `train` program. Solid arrows indicate required input/output, while dashed arrows indicate optional input/output.

The training file is a comma-separated text file that contains feature vectors and labels for a collection of samples (one per line). The first line of this file begins with "#" and specifies the contents of each column.[3] The first column will be the sample ID string, the second column contains the class (0 or 1), and the remaining columns correspond to features. Subsequent lines contain the actual sample ID, class, and feature vector for each of the data samples. An example training file looks like this:

```
#ID, Class, Feature1, Feature2, Feature3
s9173, 0, .327898, 1.2569, 3
s221, 0, .36726, 2.17864, 15
.
.
.
s17865, 1, .75897, 2.3269, 4
```

Figure 2 – An example input feature file

Training a forest via `train` will generate three output files.

The *Random Forest model file* that is output by the `train` program is a binary file containing the forest that was learned. This model file can serve as input to the `classify`, `proximity`, and `forestcat` binaries.

The log file (default name: *`modelfile`*`.train_log`) contains:

[3] This is *not* a comment line and should not be treated as such.

- messages regarding the training parameters that were used to train the Random Forest model,
- out-of-bag (OOB) estimates of Breiman's strength and correlation,
- the average accuracy of individual trees and the accuracy of the entire ensemble,
- descriptive statistics for individual trees including depth, number of nodes, and OOB false positive and true positive rates,
- the total training time of a particular forest.

The *OOB results file* (a.k.a., *OOB score file*; default name: `modelfile.oob_results`) contains the trained forest's OOB score for each sample. An example OOB score file is shown in Figure 3.

```
#ID, Class, True Class, Score, Count
0, 0, 0, -1, 90
1, 0, 0, -0.97, 89
.
.
.
15794, 1, 1, 0.37, 87
```

Figure 3 – An example OOB results file

As in the training file, the first line in the OOB score file begins with "#" and details the contents of the comma-separated columns. The first column is the sample index, in which 15794 denotes the 15795th input sample found in the training file (in the case of multiple training files it is the 15795th sample read in by the train program, where each file is read in the order of its specification on the command line). The second and third columns are the predicted class and true class of the sample, respectively. The fourth column contains the OOB score, which ranges between −1 (class 0) and 1 (class 1). The final column contains the count of all trees for whom a sample is out-of-bag.

If specified, the optional OOB sample indices file and OOB signed node indices file will be saved as `modelfile.oob_samples` and `modelfile.oob_signed_node_indices`, respectively. Row *i* of the OOB sample indices file lists the indices of the OOB training samples for tree *i*. This information is most useful for diagnostic purposes. Row *i* of the OOB signed node indices file lists the index of the leaf node that each OOB sample falls into for tree *i*. The sign of each index indicates whether the OOB sample was correctly (positive) or incorrectly (negative) classified by the tree. Consider, for example, the OOB sample and signed node indices files shown in Figures 4 and 5.

```
1 3 8
2 6 7
```

Figure 4 – An example OOB sample indices file

```
33 -14 72
85 377 -21
```

Figure 5 – An example OOB signed node indices file

In Figure 4, tree 1's OOB samples are samples 1, 3, and 8, and tree 2's OOB samples are 2, 6, and 7. Sample 1 falls in leaf node 33 of tree 1 and is correctly classified, sample 3 falls in leaf node 14 of tree 1 and is incorrectly classified, …, sample 7 falls in leaf node 21 of tree 2 and is incorrectly classified. Since most users do not require this level of detail, `train` does not produce these data files by default. However, if this output is desired, the user must specify the `-printOOBDetails` flag on the command line.

`train` Parameters

The `train` program is quite flexible and gives the user direct control over many training options. Here, we briefly describe each command line option.

Parameter	Meaning	Default
`-i` <infile> [infile ...]	Name of one or more input feature file(s) for training.	
`-s` <modelfile>	Name of the output model file.	
[`-f` <feature name list>]	Comma-separated names of features to use for training.	use all features
[`-x` <feature name list>]	Comma-separated names of features to exclude for training. One can only specify either `-f` or `-x`, not both.	use all features
[`-n` <forestsize>]	Number of trees in the forest.	500
[`-seed` <int>]	Seed for random number generator.	current time
[`-nodeTransform` <int>|None|DRF]	Transformation applied to the data at each tree node. "DRF" specifies Discriminant Random Forest, while "None" specifies Breiman's Random Forest.	DRF
[`-threshOpt` <int>|Misclassification|Gini|Theoretic]	Thresholding methodology used for node splitting. DRF uses the theoretic threshold, while Breiman's Random Forest uses Gini impurity reduction.	Theoretic
[`-dim` <int>]	Number of features to consider at each node for splitting, i.e., split dimensionality.	sqrt(number of features)
[`-depth` <max_depth>]	Maximum depth to which trees will be grown	no max depth restriction
[`-size` <max_size>]	Stop training trees once they reach <max size> nodes.	no max size restriction
[`-stop` <number>[%]]	Stop splitting nodes once node data is less than <number> (may be specified as a percentage with respect to overall training data size via "%").	no minimum data sample restriction
[`-printOOBDetails`]	Print OOB sample indices and OOB signed node indices files to *`modelfile`*`.oob_samples` and *`modelfile`*`.oob_signed_node_indices`.	flag unset (files not generated)
[`-no_bagging`]	Do not perform bagging, i.e., train each tree on all of the training samples.	flag unset (bagging performed)

[-balanced_training]	During bagging, sample class 0 and class 1 equally.	flag unset (classes sampled proportionally)
[-bagsize <number of samples to bag>]	Specifies the number of samples in the bagged training set for each tree.	number of training samples
[-cost_per_neg_example <cost per negative example>]	When -threshOpt is *Misclassification*, this is a penalty on misclassification of the negative class. The penalty on misclassification of the positive class is always 1, so if cost per negative example > 1, false alarms are penalized more heavily than missed detections and vice versa.	1
[-prob_of_neg_example_for_bagging <prior prob. of negative example for bagging>]	The probability of sampling a negative example for bagging. The probability for sampling a positive example is one minus this number.	proportion of negative samples in the training data
[-normmode <int> |0: none 1: mean/var 2: median/MAD]	Prior to training, each feature is either not normalized (0: none), normalized by subtracting the mean and dividing by the standard deviation computed over the entire input data set (1: mean/var), or normalized by subtracting the median and dividing by 1.482*Median_Absolute_Deviation (2: median/MAD).	mean/var
[-m <classification method 0: left/right 1: majority>]	Predicted class for a sample falling in a leaf node is determined either by which child node the leaf node is (0: left/right) or the majority class of training samples falling in the leaf node (1: majority).	left/right
[-sw <sample weight file>]	The sample weight file is a text file containing the probability for sampling each input training data sample (one per line) during bagging. This allows the user complete control to over/under sample any specific training sample.	uniform sampling
[-log <output log file>]	Output log file name.	*modelfile*.train_log

classify

Once a Random Forest is trained, we can use it to classify new data samples using the `classify` executable. In a nutshell, `classify` takes as input a trained model file and an input feature file (of the same format as the training file used by `train`), and outputs a score file in a similar format to the OOB results files, but absent the column of OOB counts. Figure 6 shows the box diagram for the required and optional inputs and outputs of `classify`.

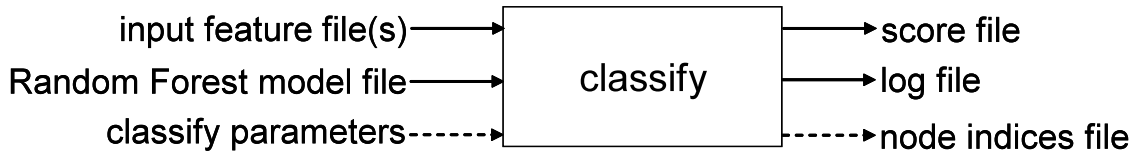

Figure 6 – Box diagram for the inputs and outputs of `classify`. Solid arrows indicate required input/output, while dashed arrows indicate optional input/output.

The log file of `classify` (default name: *`modelfile`*`.classify_log`) prints out parameter settings, status messages, and total run time.

The optional node indices file contains the indices of the leaf nodes that a testing sample falls in for each tree in the forest with a sign indicating correct (positive) or incorrect (negative) classification. The first column contains the test sample number, and each ensuing column corresponds to a separate tree in the forest. Each row corresponds to a separate data sample. The example node indices file in Figure 7 shows that test sample 0 falls into leaf node 6 of tree 1 and was correctly classified, test sample 0 falls into leaf node 28 of tree 2 and was incorrectly classified, …, and test sample 1 falls into leaf node 290 of tree 3 and was incorrectly classified.

```
0 6 -28 396
1 17 39 -290
```

Figure 7 – An example node indices file.

As mentioned above, the score file generated by `classify` is identical to the OOB results file generated by `train` with the exception of the last column. In particular, the first row specifies the contents of the comma-separated columns, and it starts with "#". The columns are:

- Sample ID – A sample's identifier
- Class – The class predicted by the Random Forest. This is 1 if the score is greater than the *decision threshold*, which defaults to 0, but can be specified using the `-t` switch.
- True Class – The true class of the data sample
- Score – The overall score resulting from tallying the votes of the individual trees. This ranges from −1 to 1. The user can weight the votes for the negative class using the `-cost_per_neg_vote` switch. In particular, the score is given by:

$$score = 2\frac{N_{positiveVotes}}{N_{positiveVotes} + cpnv * N_{negativeVotes}} - 1 \qquad (1)$$

  where $N_{positiveVotes}$ and $N_{negativeVotes}$ are the number of trees in the forest that classified this sample as positive and negative respectively, and *cpnv* is the cost per negative vote specified by the user.

```
#ID, Class, True Class, Score
0, 0, 0, -1
1, 0, 0, -0.95
.
.
.
91792, 1, 1, 0.82
```

Figure 8 – An example score file

`classify` Parameters

`classify` accepts the following command line arguments:

Parameter	Meaning	Default
`-i` <infile> [infile ...]	Name of one or more input feature file(s) containing samples to classify.	
`-s` <modelfile>	Input Random Forest model file.	
`-o` <scorefile>	Output score file	
[`-t` <threshold>]	Optional decision threshold	0.0
[`-cost_per_neg_vote` <cost per negative vote>]	Cost per negative vote	1.0
[`-m` <classification method 0: left/right 1: majority>]	Predicted class for a sample falling in a leaf node is determined either by which child node the leaf node is (0: left/right) or the majority class of training samples falling in the leaf node (1: majority).	left/right
[`-on` <output node indices file>]	Output node indices file	Unspecified (file not generated)
[`-log` <output log file>]	Output log file name.	*`modelfile`*`.classify_log`

`proximity`

Breiman's proximity metric measures the degree to which two different data samples are close to each other, i.e., fall in the same leaf nodes in a forest. The proximity of data samples t_1 and t_2 in forest f is determined by the total number of times t_1 and t_2 fall in the same leaf node during classification divided by the total number of trees in f. Given a Random Forest model, the `proximity` program computes proximities between all data samples contained in Set 1 input feature file(s) and those contained in Set 2 input feature file(s). Figure 9 depicts the required (solid arrows) and optional (dashed arrows) inputs and outputs for `proximity`.

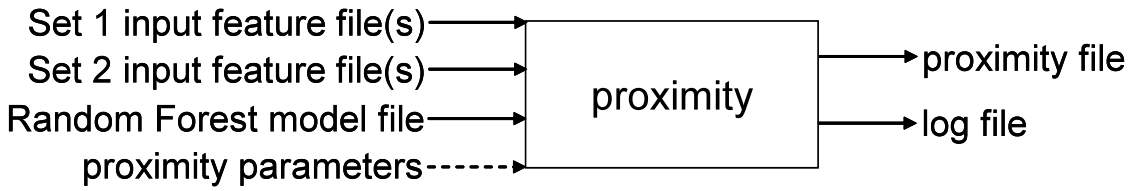

Figure 9 – Box diagram for the inputs and outputs of the `proximity` program. Solid arrows indicate required input/output, while dashed arrows indicate optional input/output.

The output proximity file contains all of the computed non-zero proximities between all pairs of Set 1 and Set 2 data samples. The file has three comma-separated columns whose contents are specified by the first row. The header row begins with "#", followed by the Set 1 input sample index (ID1), the Set 2 input sample index (ID2), and the computed proximity between the two data samples (Proximity). The remaining rows contain the sample pair indices and computed proximity. Figure 10 is an example proximity file

showing that sample 0 of Set 1 has a 0.036 computed proximity with sample 0 of Set 2, …, sample 8 of Set 1 has a 0.004 computed proximity with sample 1 of Set 2.

```
#ID1, ID2, Proximity
0, 0, 0.036
0, 1, 0.028
0, 4, 0.008
0, 8, 0.008
0, 2, 0.004
1, 0, 0.064
2, 0, 0.088
3, 0, 0.308
4, 0, 0.204
5, 0, 0.06
6, 0, 0.172
7, 0, 0.076
8, 0, 0.056
8, 1, 0.004
```

Figure 10 – An example proximity file

`proximity` Parameters

`proximity` accepts the following command-line arguments:

Parameter	Meaning	Default
`-i1` <inputfile from set 1> [inputfile ...]	Set 1 input feature file(s)	
`-i2` <inputfile from set 2> [inputfile ...]	Set 2 input feature file(s)	
`-s` <modelfile>	Random Forest model	
`-o` <proximityfile>	Output proximity file	
[`-topN` <topN proximities to output>]	For each sample in set one, print only the top N most proximal samples in set two	print all proximities
[`-m` <classification method 0: left/right 1: majority>]	Predicted class for a sample falling in a leaf node is determined either by which child node the leaf node is (0: left/right) or the majority class of training samples falling in the leaf node (1: majority).	left/right
[`-log` <output log file>]	Output log file name.	*`modelfile`*`.proximity_log`

`forestcat`

Sometimes it is useful to concatenate two smaller Random Forests to form one larger Random Forest. `forestcat` is the program that implements this capability. Additionally, `forestcat` can form a smaller forest from a subset of the trees in a larger forest. This capability enables simple parallelization of Random Forest training. For example, a large 1000-tree Random Forest can be trained by first training 100 10-tree forests on separate CPUs with different random seeds. `forestcat` can then be used to concatenate these 100 10-tree forests to form a 1000-tree forest.

`forestcat` Parameters

`forestcat` takes the following command-line arguments:

Parameter	Meaning	Default
`-i` <input_forestfile> [input_forestfile ...]	Two or more input forest model files to concatenate, or only one input forest model file to subset	
`-s` <output_forestfile>	The resulting concatenated or subsetted output forest model file.	
[`-xs` <cut_start>]	The index of the first tree in the first input forest to subset	first tree in the first forest
[`-xe` <cut_end>]	The index of the last tree in first input forest to subset	last tree in first forest
[`-log` <output log file>]	Output log file name.	*output_forestfile*.forestcat_log

If either `-xs` or `-xe` are specified, then forest subsetting is performed on the first specified forest; otherwise, forest concatenation is performed.

`ComputeROC`

`ComputeROC` computes Receiver Operating Characteristic (ROC) curves from the input score file(s) in the score file format (generated by `classify`). The resulting output ROC curve file is a text file with three comma-separated columns. The first column contains false positive rates, followed by the true positive rates, and the last column contains decision thresholds that were used to generate the corresponding false positive and true positive rates. A simple example of an ROC output file is shown below:

```
0.0, 0.0, 1.000001
0.1, 0.3, 0.75
0.3, 0.9, 0.0
0.5, 0.93, -0.5
0.7, 0.98, -0.8
1.0, 1.0, -1.00001
```

Figure 11 – An example ROC curve file generated by `ComputeROC`

`ComputeROC` Parameters

`ComputeROC` takes the following command-line arguments:

Parameter	Meaning
`-i` <inputfile> [-i <inputfile> ...]	Input score file
`-o` <outputfile>	Output ROC curve file

ComputeCost

In many applications, we are not only interested in quantifying the error rates, but also the expected cost incurred by a classification system. This metric is given by:

$$EC = p(+)\cdot(1-DR)\cdot c(miss) + p(-)\cdot FAR\cdot c(falsealarm) \quad (2)$$

where DR is the detection rate, $p(\cdot)$ is the prior probability for each class, and $c(\cdot)$ is the cost for each type of error. To enable visualization of this metric, Drummond and Holte developed "cost curves" that express expected cost as a function of the class priors and costs [Drummond2000]. Specifically, cost curves plot the expected cost (normalized by its maximum value) versus the probability cost function (PCF), which is given by:

$$PCF = \frac{p(+)\cdot c(miss)}{p(+)\cdot c(miss) + p(-)\cdot c(falsealarm)} \quad (3)$$

Assuming equal priors, PCF is small when the cost for false alarms is large relative to that of missed detections.

The `ComputeCost` program is used to compute cost curves from input ROC curve files like those generated by `ComputeROC`. The resulting cost curve file is a space-separated text file containing two columns. The first column contains the PCF points on the cost curve, and the second column contains the corresponding normalized expected cost. A sample cost curve file is shown below:

```
0.0000000000000000 0.0000000000000000
0.1458783862392024 0.0166554550450427
0.3596129796436564 0.0285333571560918
0.7908049827420335 0.0235957884370535
0.9806643972094288 0.0083922021734594
1.0000000000000000 0.0000000000000000
```

Figure 12 – An example cost curve file generated by `ComputeCost`

`ComputeCost` Parameters

`ComputeCost` takes the following command-line arguments:

Parameter	Meaning
`-i` <inputfile>	Input ROC curve file
`-o` <outputfile>	Output cost curve file

3. Perl Scripts

In addition to the C/C++ binaries, the Random Forest Toolbox includes several useful wrapper Perl scripts described briefly below. These scripts can be found in the `<toolbox_base>/Scripts` directory.

ComputeROC-oob.pl

This script is a wrapper for `ComputeROC` when it is used to compute the ROC curves from an OOB results file. `ComputeROC` can be applied directly to the OOB results file, but it is not equipped to properly handle cases where a training sample is in-bag for all trees in the forest. For most training sets the chance of this happening is quite small, but on very small training sets (<10 samples) this can become an issue. `ComputeROC-oob.pl` discards the scores from the training samples that were in-bag for all trees, computes the ROC curve from the remaining scores, and prints out a warning message to alert the user that there were such training samples present. `ComputeROC-oob.pl` takes the following command-line arguments:

Parameter	Meaning
`-help`	Prints usage message
`-verbose` <int>	Prints chatty info for debugging
`-prog` <string>	ComputeROC binary executable file
`-tmpdir` <string>	Directory for placing temporary files
`-i` <string>	Input OOB results file(s)
`-o` <string>	Output ROC curve file

statsFromROC.pl

This script computes performance statistics on an input ROC curve. The statistics include: the equal error rate (i.e., EER, the rate of equal false alarms and missed detections), the false alarm rate at a user-specified or default detection rate, the detection rate at a user-specified or default false alarm rate, and the area under the ROC curve (AUC) over a specified false alarm rate interval (which defaults to the interval [0.0, 1.0]).

`statsFromROC.pl` accepts the following command-line arguments:

Parameter	Meaning	Default
`-help`	Print usage message	
`-roc` <string>	Input ROC curve file	
`-det` <float>	Detection Rate for Computing False Alarms	0.95
`-far` <float>	False Alarm Rate for Computing Detection Rate	0.01
`-verbose`	Print wordy outputs	
`-as` <float>	Starting false alarm rate for AUC computation	0.0
`-ae` <float>	Ending false alarm rate for AUC computation	1.0
`-noInterp`	Do not interpolate when computing AUC	

empirical-ROC-Bands.pl

`empirical-ROC-Bands.pl` takes two or more input ROC curve files and computes empirical percentile bands: the $(C/2)^{th}$, the 50^{th}, and the $(100-C/2)^{th}$ percentile bands, where C is a user-specified "confidence" level. The approach for computing percentiles is consistent with the "vertical averaging" approach for averaging multiple ROC curves described in [Fawcett2006]. This approach steps through every false alarm rate found in the input ROC curves, and computes the desired percentiles from the corresponding detection rates (interpolated if needed).

`empirical-ROC-Bands.pl` accepts the following command-line arguments:

Parameter	Meaning	Default
`-help`	Print usage message	
`-debug` <int>	chatty info on stderr	
`-i` <string>	Input ROC files (must specify 2+ ROC files)	
`-confidence` <float>	Confidence level	0.05, which computes the 2.5^{th}, 50^{th} , and 97.5^{th} percentile bands
`-outDir` <string>	output directory	
`-outName` <string>	basename for all output ROC files	
`-doInterpAtExt`	Do interpolation at extreme FAR (FAR=0.0 or FAR=1.0)	

This script produces three output ROC curve files corresponding to the $(C/2)^{th}$, the 50^{th}, and the $(100-C/2)^{th}$ percentile bands in the output directory: *`outName_(C/2)`*`_percentile.roc`, *`outName`*`_50_percentile.roc`, and *`outName_(100-C/2)`*`_percentile.roc`.

runRFConfBands.pl

This script, which resides in the `<toolbox_base>/Scripts/ConfidenceBands/` directory, automates the process of training multiple instantiations of a single type of Random Forest, each with a different initial random seed. Each forest is tested on a specified test set and corresponding ROC curves are generated. `empirical-ROC-Bands.pl` is used to generate ROC percentile bands, and Cost curves are finally computed from these bands. The final product of this script is a set of percentile bands for ROC and Cost curves in *`outDir`* that can be used for significance comparisons with other Random Forest variants. It accepts all of the same arguments that `train` accepts, which allows the user to specify the type of Random Forest desired. Additionally, it accepts the following command-line arguments:

Parameter	Meaning
`-help`	Print usage message
`-debug` <int>	chatty info on stderr
`-outDir` <string>	Input ROC files (must specify 2+ ROC files)
`-trainName` <string>	Experiment name for training files

`-classifyName` <string>	Experiment name for classify files
`-confName` <string>	Experiment name for confidence bands files
`-seed` <int>	Starting random number generator seed for experiments
`-numReseed` <int>	Number of training repeats, i.e., instantiations
`-restartLevel` <int>	Restart level 0: Restart from training all RFs 1: Restart from classifying test set and generating ROC curves 2: Restart from aggregation of ROC curves for computing percentile bands
`-confidence` <float>	Confidence level of confidence bands
`-it` <string>	Input training file(s)
`-ic` <string>	Input testing file(s)
`-trainProg` <string>	RF train program
`-classifyProg` <string>	RF classify program
`-rocProg` <string>	ComputeROC program
`-rocBandsProg` <string>	ROC Confidence program, i.e., `empirical-ROC-Bands.pl`
`-costProg` <string>	ComputeCost program

4. Random Forest Variants

In this section we describe the Random Forest algorithms that are included in the Random Forest Toolbox in further detail.

The Random Forest Algorithm

The Random Forest (RF) is a highly effective classification methodology that consists of a "forest" of decision trees that are grown independently from a training data set. A Random Forest predicts the class of a test sample by voting the individual decision tree class predictions. Specifically, the ratio of the number votes for class k to the total number of trees is a measure of the degree to which the RF believes the sample is from class k.

To train an RF, individual decision trees are trained on bootstrap samples of the original training data and added to the forest until the desired number of trees is reached. To train a single decision tree, decision nodes split the tree's training data in a breadth-first manner until all nodes are homogeneous (i.e., containing only samples from a single class) or some predefined stopping condition is reached (e.g., minimum number of data samples at a node). Each decision node partitions or "splits" its incoming samples into two sets, one of which is passed to the node's left child while the other is passed to the node's right child. One key characteristic that contributes to the power of the RF, shared among all its variants, is the random sampling of features (or feature subspaces) at each node upon which the splitting decision is based. The differences among the RF variants described below lie primarily in how they learn to split the data at each node.

RF and Gini RF

The simplest classifier in the Random Forest Toolbox is the RF, whose node splitting criteria is based solely upon the node-level misclassification rate. At each node in the decision tree, as mentioned above, data are partitioned ("split") into two sets – one set goes to the left child node, and the other goes to the right child node. Successive nodes are added until the resulting partitioned data sets are homogeneous. Each split is based upon a subset of features (of some predetermined dimension) uniformly sampled from the set of all available features. For each feature subset, the single feature and threshold that minimize the misclassification rate of the resulting data partition is computed. Thus, the separating hyperplane within a node is determined by the threshold on the feature dimension yielding the lowest misclassification rate, as given by:

$$\arg\min_{t,f} misclass(t,f) = N_{falseAlarms}(t,f) + N_{missedDetections}(t,f) \tag{4}$$

$$N_{falseAlarms}(t,f) = \sum_{x_R \in D^{R,t,f}} i(y_R = 0) \tag{5}$$

$$N_{missedDetections}(t,f) = \sum_{x_L \in D^{L,t,f}} i(y_L = 1) \tag{6}$$

where (x_R,y_R) and (x_L,y_L) are training samples (feature vector, class label) from $D^{L,t,f}$ and $D^{R,t,f}$, data passed to the right and left child nodes, respectively, as a result of splitting the

current node using threshold t on feature f. Note that the resultant separating hyperplane is normal to the best feature dimension's axis.

An alternative node splitting mechanism that is more commonly used computes the decision threshold, t, that maximizes the Gini impurity reduction of the split over every sampled feature. This maximization is formally given by:

$$\arg\max_{t,f} \Delta i(t,f) = i(D) - i\left(D^{L,t,f}\right) - i\left(D^{R,t,f}\right) \tag{7}$$

where D is the data belonging to the current node, and as in the simple RF case, $D^{L,t,f}$ and $D^{R,t,f}$ are data belonging to the left and right child nodes as a result of splitting D using threshold t on feature f. $i(D)$ is the Gini impurity of D and is in general given by:

$$i(D) = 1 - \sum_{class=k} \left(p(D_k)\right)^2 \tag{8}$$

where $p(D_k)$ is the probability of class k in D. This simplifies to the following for a two-class problem:

$$i(D) = 1 - p(D_0)^2 - p(D_1)^2 = 2p(D_0)p(D_1) \cong 2\frac{n_0}{n_0+n_1}\frac{n_1}{n_0+n_1} \tag{9}$$

where n_0 and n_1 are the number of samples in class 0 and 1 respectively.

Cost-Sensitive RF

One of our motivations in pursuing this work was the development of new classifiers that could robustly achieve high detection rates at ultra-low FARs. In many cases, false alarms are more costly than missed detections. Cost-sensitivity refers to the ability of a classifier to learn decision boundaries that preferentially learn the costlier class, i.e., boundaries that sacrifice some performance for the less costly class while enhancing the performance on the costlier class.

In a RF, there are two places where one can make cost-sensitive enhancements. First, in the bootstrap sampling of training data, one can preferentially sample the costlier class. Providing more instances of the costlier class encourages the classifier to better model that class. Our RF software includes a parameter called "Probability of Negative Example for Bagging" (*pneb*), which allows the user to specify the probability of randomly sampling a negative sample. The probability of sampling a positive sample is $1-pneb$.

The other place to modify RF for cost-sensitivity is at the node splitting mechanism. Misclassification treats errors coming from false alarms and missed detections equally, so in order to make the node splitting cost-sensitive, we replace the misclassification metric (4) with the cost-sensitive misclassification metric given in (10):

$$c = cpne * N_{falseAlarms} + (1 - cpne) * N_{missedDetections} \tag{10}$$

where *cpne* is the cost per negative example, i.e., the cost of misclassifying a negative example (false alarm). Larger values of *cpne* result in decision boundaries that make fewer false alarm mistakes. As with *pneb*, our RF code supports user-specified settings for *cpne*. Good settings for *pneb* and *cpne* can be found by optimizing an OOB performance metric (e.g., Area Under the ROC curve or expected cost) via a grid search or Amoeba search.

Discriminant RF

The Discriminant Random Forest (DRF) is a powerful variant of the classical Random Forest (RF). The novelty (as well as power) of the DRF comes from the way it performs its node-level splits in the constituent decision trees. Unlike the regular RF whose node-level split boundaries are constrained to be axis-aligned hyperplanes in feature space, the DRF split boundaries can be hyperplanes of any orientation. Its multivariate hyperplanes are derived from Discriminant Analysis techniques. These types of techniques leverage the class means and covariance matrices to determine an optimal separating hyperplane under the assumption of multivariate normality. Though this assumption rarely holds in practice, we have found these techniques to be highly effective in combination with the Random Forest paradigm.

By default, the DRF utilizes the theoretical threshold defined by the separating hyperplane to split the data. Specifically, data falling on one side of the boundary will be passed to one child node, and the remaining node data will pass to the other child (the right child node is predominantly associated with the positive class and the left one with the negative class).

5. Examples

All of the examples below can be found in the `<toolbox_base>/TestData/` directory.

Example 1: A complete cycle of `train`, `classify`, `computeROC`, and `computeCost` for a Discriminant Random Forest.

In this example, we show the complete cycle of training, testing, and computing ROC and Cost curves for a Discriminant Random Forest on the example training and testing sets (T1 and J1 respectively).

`example1.pl` trains a Discriminant Random Forest on training sample files `T1_P_0.txt` and `T1_P_100.txt` with split dimensionality 2 and 250 trees on a sample dataset using only eight of the 12 available features: F1, F2, F3, F4, F5, F6, F7, F8. The `train` command-line looks like this:

```
>> ../Forests/Random/bin/train -n 250 -seed 7361973
-nodeTransform DRF -threshOpt theoretic -printOOBDetails
-dim 2 -f "F1, F2, F3, F4, F5, F6, F7, F8" -i
InputData/T1_P_0.txt -i InputData/T1_P_100.txt -s
example1.rf
```

Note that this command-line also specifies the seed for the random number generator to be 7361973, and it also requests that the OOB sample indices and OOB node indices files to be generated (`-printOOBDetails`). The trained Discriminant Random Forest model will be saved in the file `example1.rf`.

`classify` can now take the trained forest and run test set files `J1_P_0.txt` and `J1_P_100.txt` through it using the command-line:

```
>> ../Forests/Random/bin/classify -s example1.rf -i
InputData/J1_P_0.txt -i InputData/J1_P_100.txt -o
example1.testscores
```

The output score file `example1.testscores` can then be used to compute a ROC curve file using `ComputeROC` called with the following command-line:

```
>> ../ComputeROC/bin/ComputeROC -i example1.testscores -o
example1.roc
```

This results the ROC curve file `example1.roc` which can then be used by `ComputeCost` to compute a Cost curve using the following command-line:

```
>> ../ComputeCost/bin/ComputeCost -i example1.roc -o
example1.cost
```

Example 2: `forestcat` in action

This example exercises the `forestcat` utility by first building one larger forest from two smaller forests, and then subsets the larger forest to extract the first original smaller forest. More specifically, the script `example2.pl` trains two Gini Random Forests (one with 50 trees and another with 100 trees) named `example2_first.rf` and `example2_second.rf`[4]. These two forests are then concatenated using `forestcat` with the following command-line:

```
>> ../Forests/Random/bin/forestcat -i example2_first.rf -i
example2_second.rf -s example2_combined.rf
```

`example2_combined.rf` is a 150-tree Gini Random Forest whose first 50 trees are the ones from `example2_first.rf` and whose last 100 trees are from `example2_second.rf`.

Next, we can subset and extract the first 50 trees from `example2_combined.rf` and save it to a new model file named `example2_first-copy.rf` by executing:

```
>> ../Forests/Random/bin/forestcat -i example2_combined.rf
-s example2_first-copy.rf -xs 0 -xe 49
```

`example2_first-copy.rf` should be an exact copy of the original `example2_first.rf`. The user is left to verify this by executing:

```
>> diff example2_first.rf example2_first-copy.rf
```

Finally, `example2.pl` computes three separate score files of the test set for each of the following models: `example2_first.rf` `example2_first-copy.rf` and `example2_combined.rf`.

Example 3: Using `runRFConfBands.pl` to Compute Percentile ROC/Cost Bands

In this example script (`example3.pl`), we use `runRFConfBands.pl` to train multiple instances of a Discriminant Random Forest (each differing only in their random seed), to classify a test set, and finally to compute percentile bands from their ROC and Cost curves. `example3.pl` executes the following two commands:

[4] Note, in particular, that each of these Random Forests are trained using *different* random seeds. This is important when one wants to build a larger random forest from two or more smaller random forest without having replicates of trees in the larger forest.

```
>> mkdir -p ./example3

>> ../Scripts/ConfidenceBands/runRFConfBands.pl -trainProg
../Forests/Random/bin/train -classifyProg
../Forests/Random/bin/classify -rocProg
../ComputeROC/bin/ComputeROC -rocBandsProg
../Scripts/empirical-ROC-Bands.pl -costProg
../ComputeCost/bin/ComputeCost -outDir ./example3 -trainName
T1_P_100 -classifyName J1_P_100 -confName test11reseeds
-seed 4239429 -numReseed 11 -restartLevel 0 -confidence 0.1
-it ./InputData/T1_P_0.txt -it ./InputData/T1_P_100.txt -f
"f1,f2,f3,f4,f5,f6,f7,f8" -n 5 -nodeTransform DRF -threshOpt
Theoretic -dim 2 -stop 30 -m 0 -ic ./InputData/J1_P_0.txt
-ic ./InputData/J1_P_100.txt
```

The first command creates the output directory where `runRFConfBands.pl` will save files to, and the second one calls `runRFConfBands.pl` to do the real work. When running `runRFConfBands.pl`, make sure you specify the correct locations for the binary executables: `train`, `classify`, `ComputeROC`, `ComputeCost`, and `empirical-ROC-Bands.pl`; the locations can be specified using the `-trainProg`, `-classifyProg`, `-rocProg`, `-costProg`, and `-rocBandsProg` switches respectively. `-trainName`, `-classifyName`, `-confName` are basenames for the files that `runRFConfBands.pl` will produce. In this example, these are set to "`T1_P_100`", "`J1_P_100`", and "`test11reseeds`" respectively. `runRFConfBands.pl` will thus create subdirectories of the directory `<toolbox_base>/example3/` for each instance of a forest using the name `T1_P_100.`*`randomSeedUsed`*`/`. Inside these subdirectories will be the trained random forest model, and the score files and resulting ROC curve files resulting from classifying the J1 example test set using the trained model. The score files and ROC files will be named `J1_P_100.decvals` `J1_P_100.roc` respectively.

`example3.pl` calls `runRFConfBands.pl` to train 5-tree Discriminant Random Forests with early stopping at 30 samples. 11 forests are trained on the T1 sample data set using successive random seeds starting at 4239429. `runRFConfBands.pl` then computes the 5th, 50th, and 95th percentile bands from the 11 ROC curve files and saves them to `test11reseeds_5_percentile.roc`, `test11reseeds_50_percentile.roc` `test11reseeds_95_percentile.roc` in the `<toolbox_base>/example3/` directory. Finally, Cost curves are computed from these and saved to `test11reseeds_5_percentile.cost`, `test11reseeds_50_percentile.cost` `test11reseeds_95_percentile.cost` in the `<toolbox_base>/example3/` directory.

Example 4: Train, classify, compute proximities, ROC and Cost curves, subset and concatenate Random Forest variants

`example4.pl` performs a full suite of operations for the following types of Random Forests described earlier:

- Random Forest with misclassification node splitting criteria
- Cost-Sensitive Random Forest
- Random Forest with Gini node splitting criteria
- Discriminant Random Forest
- Discriminant Random Forest with Early Stopping

This script is useful for learning the command-line arguments required for training each of the above variants. For each RF variant, `example4.pl` will (1) train the forest on the example hidden signal training data, (2) classify a test set using the trained forest, (3) compute proximities between every two data points in a pair of test sets, (4) compute ROC and Cost curves, and (5) subset the forest model into two subforests and then concatenate them to reproduce the original forest mode using `forestcat`. All output model, score, OOB, ROC curve, Cost curve files will be saved in the `example4` directory.

Random Forest with misclassification node splitting criteria

The relevant training parameters for training a basic random forest with misclassification node splitting criteria are:

```
-nodeTransform none
-threshOpt misclassification
-m 0
```

These parameters specify that no node-level feature transformation is to be applied, that the threshold used for splitting data at a node should optimize for minimum misclassification, and that the final determination of class identities should be based on whether the sample falls in the left or right child leaf node (rather than by majority class).

Cost-Sensitive Random Forest

The Cost-Sensitive Random Forest is very similar to the misclassification-based Random Forest, but in this case, there are two cost-sensitive parameters that the user may specify: the probability for sampling a negative example in bagging (*pneb*) and the cost per negative example (*cpne*). *pneb* controls how likely a negative example will be sampled in each bagged training set. This is specified using the `-prob_of_neg_example_for_bagging` switch. *cpne* controls the penalty for misclassifying a negative sample when using the misclassification node splitting criteria. This is specified using the `-cost_per_neg_example` switch. The example Cost-Sensitive Random Forest in `example4.pl` is trained with the following relevant parameters:

```
-nodeTransform none
```

```
-threshOpt misclassification
-cost_per_neg_example 10
-prob_of_neg_example_for_bagging 0.65
-m 0
```

These parameters are the same as those for the misclassification-based Random Forest, except with the cost-sensitive parameters specified.

Gini Random Forest

The Gini Random Forest is the most common variant of Random Forest described in the literature. Instead of using misclassification as the metric for determining the optimal threshold for splitting data at each node, the Gini Random Forest uses the Gini impurity reduction criterion. To specify the training of a Gini Random Forest, use the following command-line parameters:

```
-nodeTransform none
-threshOpt Gini
-m 1
```

Like the previous two random forest variants, no node-level feature transformation is to be applied, but for this classifier we use the Gini impurity reduction metric for threshold optimization, and we base the final classification on the majority class of the training data that falls into a leaf node.

Discriminant Random Forest Without and With Early Stopping

Unlike the previous Random Forest variants, the Discriminant Random Forest applies a discriminant-based transformation to the data at each node and uses the resulting decision boundary as a threshold. To specify the training of a Discriminant Random Forest, use the following command-line parameters:

```
-nodeTransform DRF
-threshOpt Theoretic
-m 0
```

This set of arguments tells the trainer to perform the DRF transformation at each node and then use the theoretic threshold defined by the computed boundary. Moreover, the final classification of a sample is based upon whether a sample falls in the left or right child leaf node. Other threshold optimization methods can also be used (e.g., misclassification or Gini), but we have found the best performance comes from using the theoretic one.

To train a Discriminant Random Forest with early stopping, simply specify a forest stopping criteria. Typically, we set a threshold for the minimum number of data samples at a node required for splitting. The parameter settings below are for training a Discriminant Random Forest with an early stopping criteria of 30 mininum data samples at a node:

```
-nodeTransform DRF
-threshOpt Theoretic
-m 0
-stop 30
```

6. Benchmark Studies

In this section, we show some experimental results obtained by applying the Random Forest Toolbox on two interesting datasets: Hidden Signal Detection and MAGIC Gamma Telescope.

Hidden Signal Detection Dataset

The goal in the Hidden Signal Detection application is to detect the presence of an embedded signal. The Hidden Signal Detection application is an example of the type of real-world problems that we developed our Random Forest algorithms to address: an application requiring an acceptable detection rate with ultra-low false alarm rates. Hence, the ROC curves that we plot will have the false alarm axis in log scale so as to better see performance differences at low false alarm rates (this also applies for the MAGIC Gamma Telescope application in the next subsection). The data for these experiments are composed of two separate sets. The training data set consists of 7931 negative class samples (i.e., no embedded signal) along with 7869 positive class samples (i.e., signals embedded with 100% signal strength). For testing, we have another set consisting of 179,527 negative class samples and 9,426 positive class samples with 100% embedded signal strength. The larger number of negative samples in the test set allows us to see performance differences at false alarm rates as low as 5.57×10^{-6}. All data samples consist of eight continuous-valued features useful for detecting the presence of embedded signals.

In this subsection, we compare the performance of all the major Random Forest variants supported by the toolbox: Random Forest with misclassification-based node splitting (Misclass RF), Cost-Sensitive Random Forest (CS-RF), Random Forest with Gini impurity-based node splitting (Gini RF), Discriminant Random Forest (DRF), and Discriminant Random Forest with early stopping (DRF EarlyStop30). For each RF variant, 101 forests were trained and tested using different random seeds. Based upon the resulting ROC curves, a "median" ROC and corresponding upper and lower confidence limits were computed for each methodology using a variant of the vertical averaging approach described in [Fawcett2006]. Specifically, for each false alarm rate (FAR) value, the 101 corresponding detection rates were ranked, and their median detection rate was computed along with their 97.5^{th} and 2.5^{th} percentiles. Using this data, the median ROC (i.e., 50^{th} percentile), the 97.5^{th} , and 2.5^{th} percentile bands for best split dimension of each RF variant were computed and are shown in Figure 13. Each RF variant consists of 1000 trees.

Figure 13 shows a progression of RF variant performance that we have observed for other datasets as well. First, the Misclass RF has always been the laggard among the RF variants with respect to detection performance at low false alarm rates[5]. The CS-RF

[5] Anecdotally, the Misclass RF's performance is still better than other classification methods (e.g., support vector machines and neural networks) applied to this problem which speaks to the power of Random Forest based methods.

system in this case is simply a Misclass RF with a bias toward oversampling of the negative class which leads to a detection improvement for false alarm rates less than $3x10^{-4}$. This particular CS-RF system uses a 95% probability of negative example for bagging which emphasizes the learning of the negative class. At the other end of the ROC curve (high false alarm rates), CS-RF does not outperform Misclass-RF which is as expected since this CS-RF is not focused on modeling the positive class better than the negative class. If one desired to have ultra low miss rates instead of ultra low false alarm rates, setting the probability of sampling a negative example for bagging to some low probability would likely work.

Gini RF provides similar detection performance as the CS-RF at the very low false alarm rate region, but for higher false alarm rates it is much better (e.g., for false alarm rates greater than $3x10^{-4}$). Our newly developed DRF, which is much different than the three RF variants discussed above, shows evidence of detection superiority for false alarm rates below $2x10^{-4}$. However, with early stopping (i.e., stop training when the nodes contain fewer than some specified number of points – in this case, 30) applied to DRF, the superiority in detection performance is statistically significant for false alarm rates less than $2x10^{-3}$ where the DRF EarlyStop30 percentile bands do not overlap those of any other system.

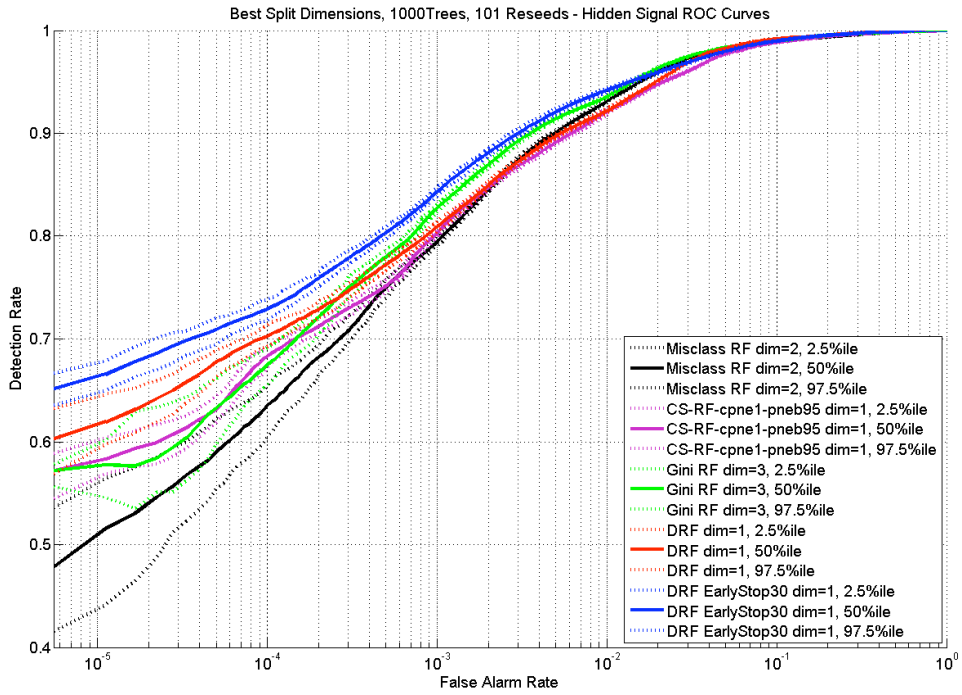

Figure 13 – ROC percentile bands (2.5th, 50th, and 97.5th percentiles) of the major RF variants supported by the toolbox applied to the Hidden Signal Detection dataset. Results are shown using the best split dimensionality for each RF variant. In the case of Cost-Sensitive RF (CS-RF) the cost per negative example is 1.0 and the probability of negative example for bagging is 95%.

In Figure 14, we plot the percentage improvement in the median ROC curves presented in Figure 13 with respect to that of the Gini RF system over the entire range of false alarms. This immediately shows the range of false alarms over which any RF variant is superior or inferior to the Gini RF. At ultra-low false alarm rates ($< 10^{-4}$), the CS-RF, DRF, and DRF EarlyStop30 all give better performance than Gini RF. However, only DRF EarlyStop30 demonstrates gains in low false alarm rates while preserving detection performance at higher false alarm rates compared to the Gini RF.

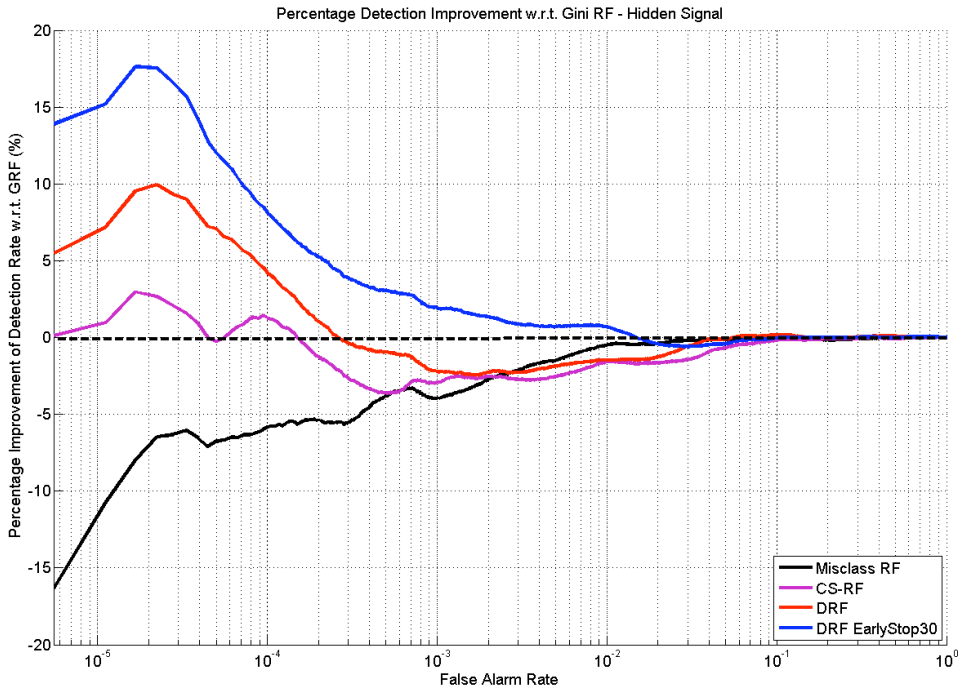

Figure 14 – Percentage improvement in median detection rates with respect to that of the Gini Random Forest on the Hidden Signal Detection dataset.

Another way to compare performance is to use the cost curves that we described in Section 2. Cost curves plot the normalized expected cost versus the probability cost function (PCF). Given a fixed prior on the positive and negative classes, lower values of PCF correspond to cases where the cost of false alarms is greater than the cost of missed detections. In Figure 15, we plot the cost curves corresponding to the ROC percentile bands displayed in Figure 13. Assuming equal priors, PCF values less than 10^{-3} correspond to the case that false alarms cost more than 100 times more than missed detections. With these cost settings, Figure 15 paints a very similar story as the ultra-low false alarm rate comparisons in Figure 13. In particular, the rankings with respect to achieving low normalized expected cost of the 5 systems is consistent – Misclass RF is the worst, Gini RF and CS-RF are better, DRF is better than these but not significantly, and DRF EarlyStop30 significantly outperforms all of the variants.

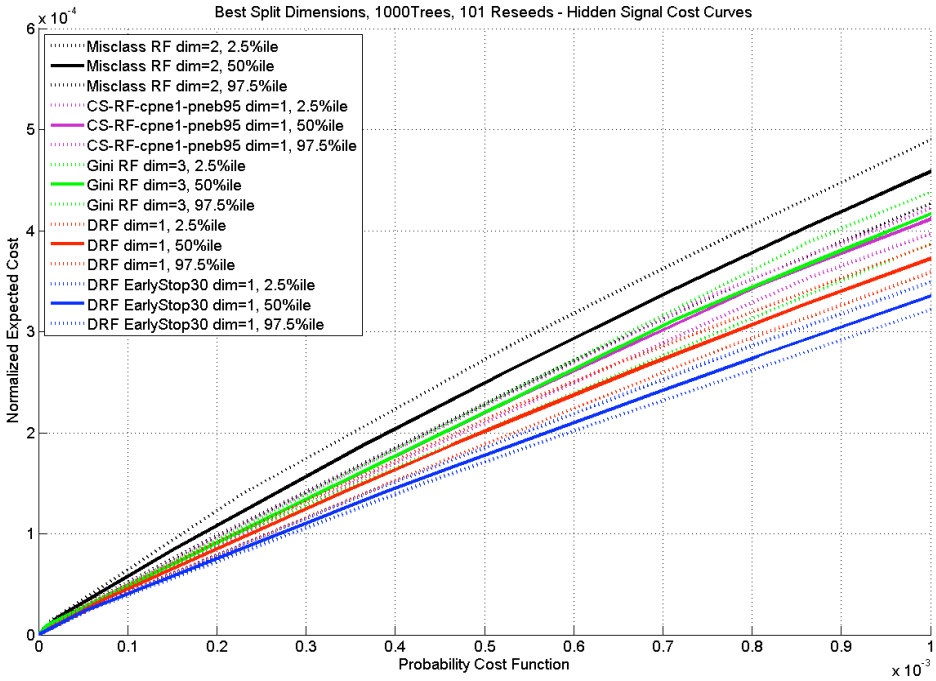

Figure 15 – Cost curve percentile bands (2.5th, 50th, and 97.5th percentiles) of the major RF variants supported by the toolbox applied to the Hidden Signal Detection dataset. These cost curves correspond to the ROC curves displayed in Figure 13.

Table 1 compares the RF variants in terms of physical characteristics of their trained forests on the Hidden Signal dataset. It shows each variant's average tree depth, average number of nodes per tree, the ratio of the average number of nodes to depth, average training time per tree, and average memory usage per tree. Interesting observations can be made about each of the RF variants based on these statistics. First, with regard to tree geometry, the Misclass RF tends to learn rather tall and skinny[6] trees. The CS-RF learns shorter and skinnier trees compared to Misclass RF. Gini-RF learns shorter trees that are denser/bushier than the Misclass RF. The DRF forests are even shorter and bushier than the rest. This presumably can be attributed to the stronger data separation at each node provided by its discriminant-based decision boundary, which precludes the need for long chains of successive node splits to generate equivalent separations. Second, in terms of training time and memory consumption per tree, Gini RF trains the fastest and has nearly the lowest footprint. Misclass RF trains much slower and takes about twice as much memory per tree than Gini RF. The CS-RF trains at nearly the same speed and takes about the same amount of memory as Gini RF. DRF trains the slowest which is not surprising given the type of threshold computation performed at each node, but with early stopping the training time is nearly on par with Gini RF. Memory usage for both DRF variants is larger than Gini RF, but in the case of DRF EarlyStop30, only twice as much memory is used per tree compared to Gini RF (11,313 bytes versus 6,711 bytes).

[6] Skinny is the opposite of dense or bushy and can be loosely characterized by the average number of nodes to depth ratio. Smaller values indicate skinniness, while larger values indicate denseness or bushiness.

RF Variant	Average Tree Depth	Average # Nodes per Tree	Average Node to Depth Ratio	Training Time per Tree (s)	Average Memory Usage per Tree (bytes)
Misclass RF	101.9	815.4	8.0	0.16	13046
Gini RF	22.5	419.4	18.7	0.06	6711
CS-RF	50.9	362.6	7.1	0.08	5802
DRF	17.4	1308.9	75.3	0.21	31413
DRF EarlyStop30	13.7	471.4	34.3	0.09	11313

Table 1 – Statistics of representative forests for each of the major RF variants trained on the Hidden Signal Detection dataset. All forests were trained on a MacPro 2 x 3 GHz Quad-Core Intel Xeon machine.

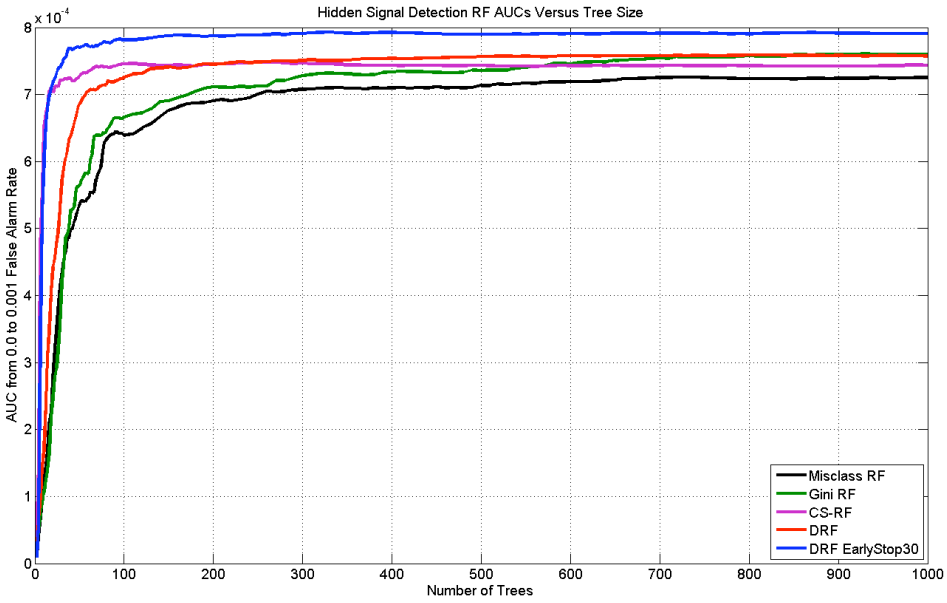

Figure 16 – The area under the ROC curve (AUC) for very low false alarm rates between 0.0 and 0.001 for representative forests of each RF variant versus the number of trees in the forest trained and tested on the Hidden Signal Detection dataset.

It is important to keep in mind that the speed and memory metrics in Table 1are computed on a per tree basis. The number of trees required to achieve equal detection performance will ultimately dictate which variant has the best overall forest speed and memory footprint. In Figure 16 we plot the area under the ROC curve (AUC) over the low false alarm rates of interest for this application (i.e., 0 to 10^{-3}) versus the number of trees in the forest for the representative RF variants used to generate Table 1. When comparing DRF EarlyStop30 to Gini RF, it takes about 25 trees for DRF EarlyStop30 to achieve an AUC of 0.0007 (70% of maximum possible AUC from 0 to 10^{-3} false alarm rates), while the Gini RF requires about 150 trees. This means that the total training time required for training a Gini RF of 150 trees is about 8.4 seconds compared to 2.2 seconds required to train a DRF EarlyStop30 of 25 trees. Similarly, the DRF EarlyStop30 of 25 trees takes about 283 Kbytes, while the Gini RF of 150 trees takes about 1,007 Kbytes. Thus, from both the detection performance, and speed/memory requirements standpoint,

the DRF algorithm with early stopping clearly outperforms all the other variants on the Hidden Signal Detection dataset.

MAGIC Gamma Telescope Dataset

The MAGIC Gamma Telescope dataset, available via the UCI Machine Learning Repository [Asuncion2008], is a two-class dataset used for differentiating simulated events observed by a Gamma-ray telescope. The goal is to classify foreground events from background events using ten continuous-valued image processing features. This dataset consists of 12,332 foreground (positive class) events and 6,688 background (negative class) events. The first 2/3 of each class is used for training, while the last 1/3 is used for testing in a manner consistent with the seminal benchmark paper published by the authors of this dataset [Bock2004]. In fact, Bock et. al. in [Bock2004] tested the Gini Random Forest's performance on this dataset and found that a split dimension of 3 was optimal and that 50 trees was sufficient for achieving good results on this dataset. As in the Hidden Signal application, this application also puts a premium on ultra-low false alarm rates according to its authors. In this subsection, we repeat their experiments and compare these results to other Random Forest variants that we have developed. We also plot percentile bands to assess the statistical significance of the comparisons and find evidence that the Discriminant Random Forest leads to better detection performance, especially at false alarm rates near 10^{-2}.

Figure 17 displays the ROC percentile bands for the four RF variants tested in this subsection: Random Forest with misclassification-based node splitting (Misclass RF), Random Forest with Gini impurity-based node splitting (Gini RF), Discriminant Random Forest (DRF), and Discriminant Random Forest with early stopping (DRF EarlyStop30). A similar ranking of detection performance at low false alarm rates is observed for this dataset as in the Hidden Signal dataset. Misclass RF underperforms the rest, Gini RF is significantly better, and DRF and DRF EarlyStop30 are better than Gini RF for false alarm rates less than $5x10^{-2}$. Unlike in the Hidden Signal detection application, the separation in the ROC percentile bands of DRF EarlyStop30 is not as strong. In fact, the ROC percentile bands of DRF EarlyStop barely avoid overlapping with those of the Gini RF for false alarm rates between $7x10^{-3}$ and $2x10^{-2}$. Despite the overlap, there is still evidence that both DRF variants give a higher chance of performance improvements over Gini RF as the percentage improvements in median ROC curves with respect to Gini RF show in Figure 18. Both DRF variants lead to large relative improvements in detection rates at the lowest false alarm rates measureable on this dataset.

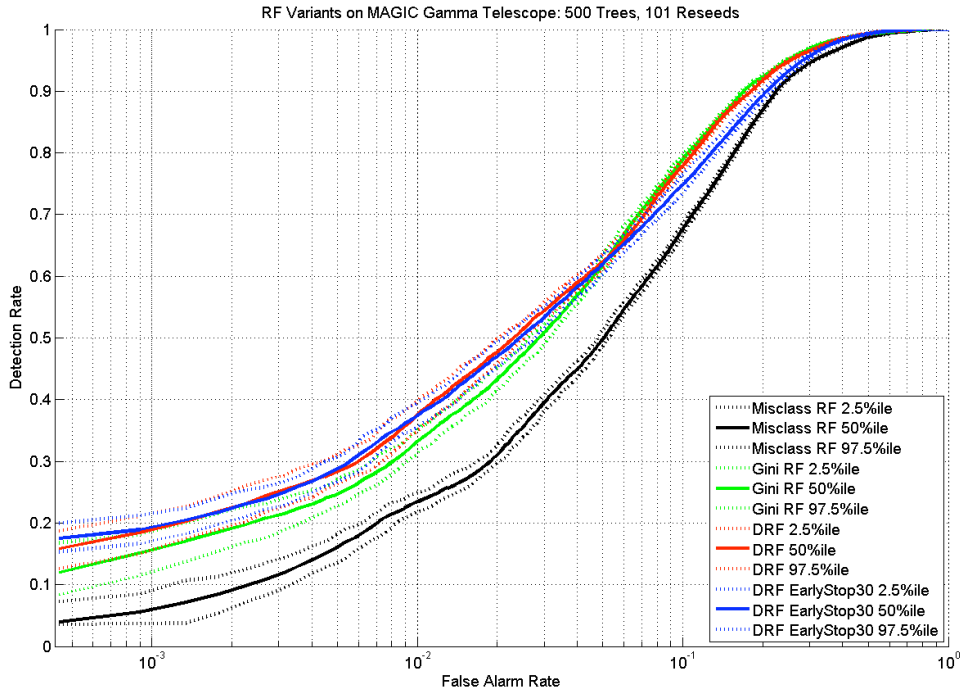

Figure 17 – ROC percentile bands (2.5th, 50th, and 97.5th percentiles) of RF variants supported by the toolbox applied to the MAGIC Gamma Telescope dataset. Results are shown using the best split dimensionality for all forests (dim=3).

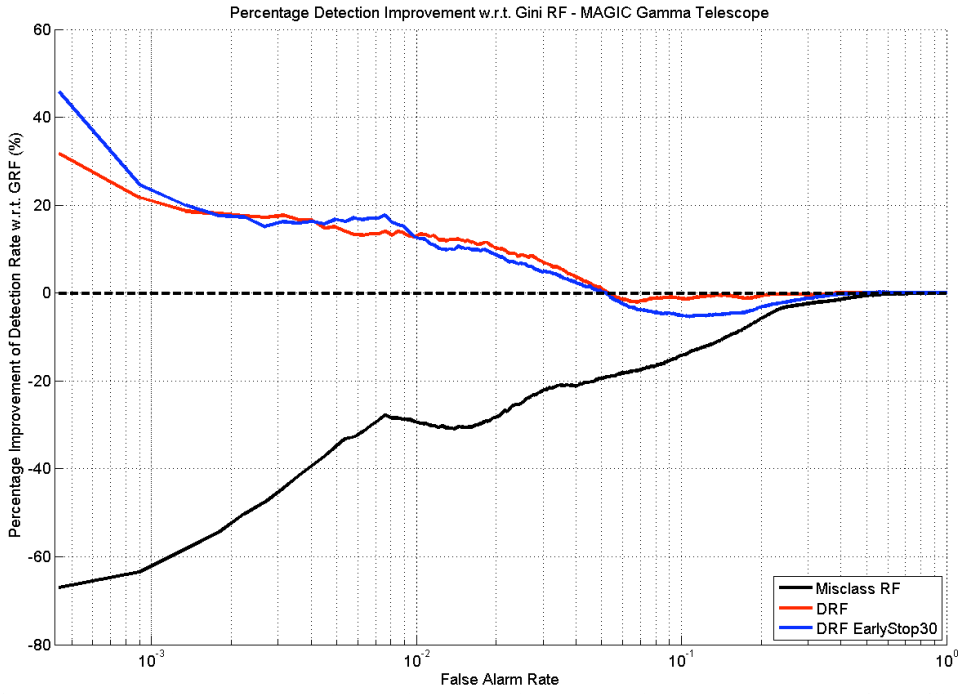

Figure 18 – Percentage improvement in median detection rates with respect to that of the Gini Random Forest on the MAGIC Gamma Telescope dataset.

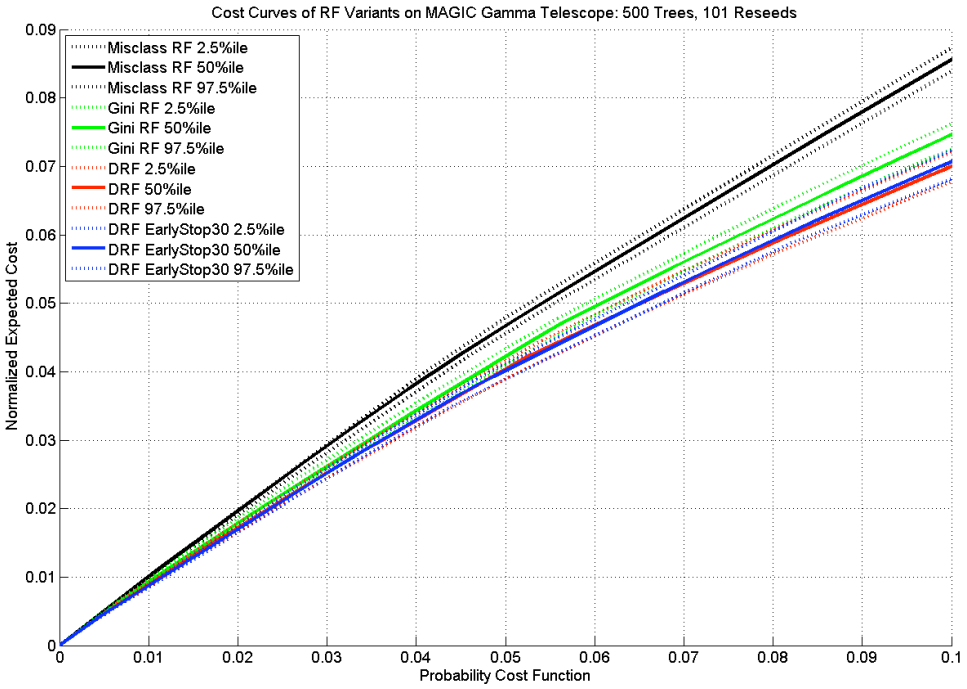

Figure 19 – Cost curve percentile bands (2.5th, 50th, and 97.5th percentiles) of RF variants supported by the toolbox applied to the MAGIC Gamma Telescope dataset. These cost curves correspond to the ROC curves displayed in Figure 17.

The cost curves in Figure 19 show the normalized expected cost of the various systems over PCF values between 0 and 0.1, which, assuming equal priors, correspond to cases where the cost for false alarms is about ten or more times greater than the cost for missed detections. In this regime, Misclass RF has significantly higher expected cost. The Gini RF has lower expected cost, while the DRF variants have even lower ones. Toward the PCF values near 0.1, the superiority of the DRF variants over the Gini RF is statistically significant as their percentile cost bands do not overlap. The bands start overlapping for PCF values less than 0.05, but the median cost curves for both DRF variants continue to lie below that of the Gini RF. This suggests that the DRF variants continue to enjoy an advantage with respect to the Gini RF when the cost for false alarms is greater than the cost for missed detections.

Table 2 contains the training statistics for each of the RF variants trained on the MAGIC Gamma Telescope dataset. Similar comments regarding the geometry of the forests when training on the Hidden Signal data can be made here as well. The Misclass RF learns extremely tall and skinny trees. Its average depth is 471.9 nodes, which is over fifteen times deeper than the next deepest tree type. Gini RF trees are much shorter than Misclass RF trees and much denser. DRF variants learn the shortest and densest trees among all RF variants. With early stopping, DRF trees are shorter and less dense compared to their non-early stopping relatives.

RF Variant	Average Tree Depth	Average # Nodes per Tree	Average Nodes to Depth Ratio	Training Time per Tree (s)	Average Memory Usage per Tree (bytes)
Misclass RF	471.9	3909.6	8.3	0.99	62553
Gini RF	31.2	2317.0	74.4	0.07	37071
DRF	16.4	4014.5	245.1	1.93	224810
DRF EarlyStop30	13.0	1011.0	78.0	0.52	56615

Table 2 – Statistics of representative forests for each of the major RF variants trained on the MAGIC Gamma Telescope dataset. All forests were trained on a MacPro 2 x 3 GHz Quad-Core Intel Xeon machine.

In terms of training time and memory usage on a per tree basis, the RF variants can be ranked in the same order on this dataset as in the Hidden Signal dataset. Gini RF trains the fastest and produces the most compact trees followed in order by DRF EarlyStop30, Misclass RF, and finally DRF. Again, it is important to note that these statistics are derived on a per tree basis, so to gauge which *forest* trains the fastest and has the lowest memory footprint, we must take into account how many trees are required to achieve a level of desired performance. Figure 20 displays the area under the ROC curve for each of the representative forests in Table 2 as a function of the number of trees in the forest. Figure 20 shows the AUC computed over the lowest false alarm rates between 0 and 0.01 which is motivated from the desire for high detection performance at the lowest false alarm rates.

Misclass RF's AUC more or less plateaus around 210 trees. Gini RF reaches this point at about 125 trees. DRF flattens out at about 100 trees, while DRF EarlyStop30 levels out at about 180 trees. It is interesting to compare DRF EarlyStop30 with Gini RF at the AUC level where Gini RF tops out (0.0025 or 25% of maximum AUC over this false alarm region). It takes the Gini RF about 125 trees to reach this level of performance, while it takes only about 25 trees for DRF EarlyStop30. In terms of overall forest training times for these systems, Gini RF forest trains faster; it takes about 8.75 seconds for the Gini RF and 13 seconds for DRF EarlyStop30. As far as memory is concerned, DRF EarlyStop30 has a smaller footprint; 1,415 Kbytes versus 4,633 Kbytes. While DRF EarlyStop30 does not train faster than a comparable Gini RF on this dataset, it is more compact which makes it more attractive for applications requiring both low false-alarm rates and small memory footprints.

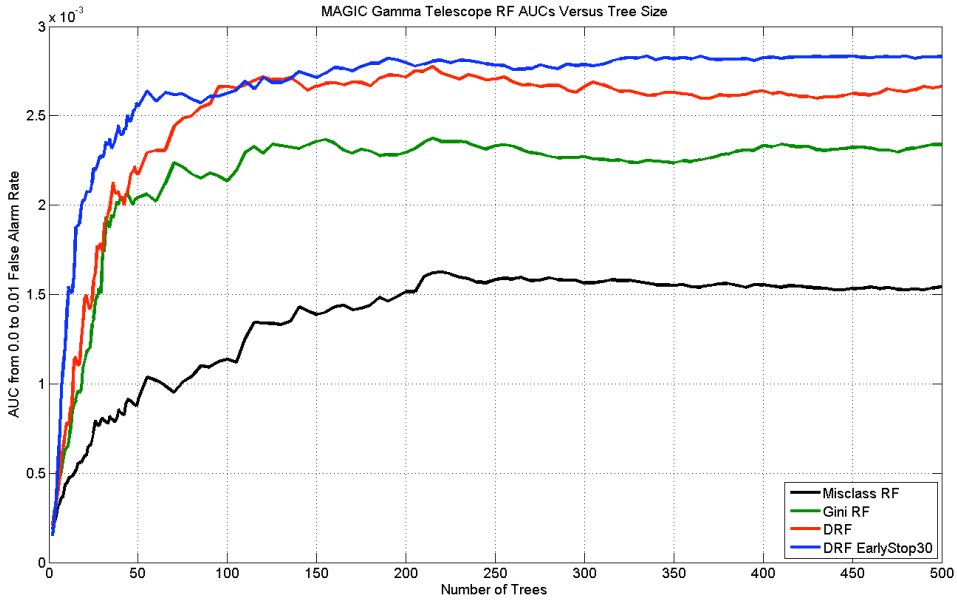

Figure 20 – The area under the ROC curve (AUC) for very low false alarm rates between 0.0 and 0.01 for representative forests of each RF variant versus the number of trees in the forest trained and tested on the MAGIC Gamma Telescope dataset.

7. References

[Asuncion2008] A. Asuncion and D.J. Newman, UCI Machine Learning Repository [http://www.ics.uci.edu/~mlearn/MLRepository.html]. Irvine, CA: University of California, School of Information and Computer Science, 2008.

[Breiman2001] L. Breiman, "Random Forests", Machine Learning, vol. 45, no. 1, pp. 5-32, 2001.

[Bock2004] R. K. Bock, A. Chilingarian, M. Gaug, F. Hakl, T. Hengstebeck, M. Jirina, J. Klaschka, E. Kotrc, P. Savicky, S. Towers, A. Vaiciulis, W. Wittek, "Methods for Multidimensional Event Classification: A Case Study Using Images from a Cherenkov Gama-Ray Telescope", Nuclear Instruments and Methods in Physics Research Section A, vol. 516, pp. 511-528, 2004.

[Drummond2000] C. Drummond and R. Holte, "Explicitly Representing Expected Cost: An Alternative to ROC Representation", in Proc. of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2000.

[Fawcett2006] T. Fawcett, "An Introduction to ROC Analysis," Pattern Recognition Letters, 27, pp. 861-874, 2006.

[Ho1998] T. K. Ho, "The Random Subspace Method for Constructing Decision Forests", IEEE Trans. On Pattern Analysis and Machine Intelligence, vol. 20, no. 8, pp. 832-844, 1998.

[Lemmond2008] T. D. Lemmond, A. O. Hatch, B. Y. Chen, D. A. Knapp, L. J. Hiller, M. J. Mugge, W. G. Hanley, "Discriminant Random Forests," Proceedings of the 2008 International Conference on Data Mining (DMIN'08), July 2008.

[Lemmond2009] T. D. Lemmond, B. Y. Chen, A. O. Hatch, and W. G. Hanley, "An Extended Study of Discriminant Random Forests," Annals of Information Systems Special Issue on Data Mining. *In Press*.